Class Notes — 18 June 2014
Tufts COMP 140, Chris Gregg

Detecting and Enhancing Loop-Level Parallelism

Loops: the reason we can parallelize so many things

If the compiler can figure out if a loop is parallel, it can produce parallel code that can utilize parallel processors (e.g., for a GPU, or multicore CPU)

We are going to define precisely when a loop is parallelizable, how dependence can prevent a loop from being parallel, and techniques for eliminating some types of dependencies.

From a computer architecture perspective, we can utilize all of our interesting Data Level Parallelism, Thread Level Parallelism, and even Instruction Level Parallelism if we can figure out where the dependencies are in our loops, and remove them.

We will focus on data dependencies: when an operand is written at some point and read at a later point. Example:

```
for (i=0;i<100;i++) {
      A[i+1] = A[i]+B[i]    ;
}
```

`A[2]` depends on `A[1]`, which is written in the first iteration. This is a data dependence.

In order to analyze loop-level parallelism, we need to determine whether there is a “loop-carried dependence” — i.e., whether data accesses in later iterations are dependent on data values produced in earlier iterations.

Is the following loop-level parallel?

```
for (i=999; i>=0; i=i-1)
      x[i] = x[i] + s;
```

Indeed, it is, and we can unroll it completely:

```
x[999] = x[999] + s;
x[998] = x[998] + s;
…
x[0] = x[0] + s;
```

This would be a trivial loop to parallelize on a GPU — each thread adds s to its element of the array.

In the above loop, the two uses of x[i] are dependent, but because it is within a single iteration, it is *not* loop carried.

This is actually a case where the compiler will want to attempt this loop-level parallelism recognition at the source level instead of the machine code level — the high-level loop structure

actually helps to abstract the parallelism, where a machine code set of instructions might make it less clear.

Let's take a look at the dependencies in a more complex example:

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that A, B and C are distinct, non-overlapping arrays (if they were overlapping, we would have a much harder time analyzing this!)

What are the data dependencies between the statements S1 and S2 in the loop?

Answer:

There are two difference types of dependencies:

1. `S1` uses a value computed by `S1` in an earlier iteration, since iteration `i` computes `A[i+1]`, which is then read in iteration `i+1`. The same is true of `S2` for `B[i]` and `B[i+1]`.

2. `S2` uses the value `A[i+1]` computed by `S1` in the same iteration.

The two dependencies have two different effects. We will assume only one effect happens at a time.

Because the dependence of S1 is on an earlier iteration of S1, this dependence is loop carried. We must serialize this. (boo!)

The second dependence (S2 depending on S1), however, is within the iteration, and is *not* loop carried. If this was the only dependence, multiple iterations could execute in parallel, as long as each pair of statements in each iteration were kept in order. Intra-loop dependencies are common, and we are able to parallelize them.

Are loop-carried dependencies always a dead-end for parallelism? No! Let's look at another example:

```
for (i=0; i<100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependencies between S1 and S2? Is this loop parallel? If not, can we make it parallel?

Answer: Statement S1 uses the value assigned in the previous iteration by S2, so there is a loop-carried dependence between S2 and S1. However, this can be made parallel, because the dependence is not circular — neither statement depends on itself, and only S1 depends on S2, not the other way around.

A loop is parallel if it can be written without a cycle in the dependencies: the absence of a cycle means that the dependences give a partial ordering on the statements.
We do have to make this loop conform to the parallel ordering to expose the parallelism. We make two observations:

1. There is no dependence from S1 to S2. If there were, then this would be a cycle in the dependencies, and the loop would not be parallel. Since this other dependency is absent, we can exchange the order of the statements, which won't affect the execution of S2.

2. On the first iteration of the loop, S1 depends on the value of B[0], which was computed prior to the loop.

With these two observations, we can replace the loop above with the following code:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

The dependencies are no longer loop carried, so we can now parallelize the loop! We do need to ensure that the two statements in the loop remain in order, but taken together, they are independent of all the other iterations of the loop.

————

How can we more formally analyze these loop carried dependencies? Unfortunately, the dependence information is inexact, in that it tells us that a dependency may exist, not that it does. Here's another example:

```
for (i=0; i<100; i=i+1) {
      A[i] = B[i] + C[i]; /*S1*/
      D[i] = A[i] + E[i]; /*S2*/
}
```

Even though there is a dependence with A[i], the second reference to A[i] doesn't need to be loaded—it will already be in a register, so we can use it in statement 2 after it has been loaded in statement 1, without explicitly loading A[1]. In a sense, we could re-write it in the following way, so that the order of statement S1 and S2 are completely independent:

```
for (i=0; i<100; i=i+1) {
      T = B[i] + C[i];
      A[i] = T;         /*S1*/
      D[i] = T + E[i]; /*S2*/
}
```

In reality, data dependence analysis only tells us that a reference *may* depend on another, but we would need more complex analysis to tell us that the references must be the same address. In this case it is easy, because it is a simple code block.

Sometimes, we have a loop-carried dependency in the form of a *recurrence*, which occurs when a variable is defined based on the value of that variable in an earlier iteration (often the one immediately preceding it):

```
for (i=1; i<100; i++) {
      Y[i] = Y[i-1] + Y[i];
}
```

We want to detect this for two reasons:

1. Some architectures have special support for handling recurrences
2. It might be possible to exploit instruction level parallelism

How do we find these dependencies?

We have to be careful — with pointers, it gets *really* hard — an analyzer must keep track of all references.

Even with these difficulties, we will proceed. We will first assume that array indices are *affine*. A on-dimensional array index is affine if it can be written in the following form:

$a \times i + b$, where a and b are constants, and i is the loop index variable.

To determine whether there is a dependence between two references, to the same array in a loop is equivalent to determining whether two affine functions can have the same value of different indices between the bounds of the loop. Example:

Say we have stored to an array element with index value

$a \times i + b$

and loaded from an array element with the index value

$c \times i + d$

where i is the for-loop index that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indexes, j and k, that are both within the limits of the for loop. E.g., $m \leq j \leq n, \quad m \leq k \leq n.$

2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from the same array element when indexed by $c \times k + d$. In other words,

$$a \times j + b = c \times k + d$$

In general, we can't determine this at compile time. We might not know a, b, c, and d (they could be values in other arrays, for instance).

However, luckily, most programs contain simple indexes where a, b, c, and d are all constants. We can actually come up with a test at compile time for dependence.

One test is the *greatest common divisor* (GCD) test. It is based on the following observation:

If a loop carried dependence exists, then GCD(c,a) must divide (d - b). (Recall: an integer, x, divides an integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.)

Use the GCD test to determine whether dependencies exist in the following loop:

```
for (i=0; i<100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
}
```

Answer:

Given the values a=2, b=3, c=2, and d=0, then GCG(a,c) = 2 and d - b = -3. Since 2 does not divide -3, no dependencies are possible.

We can unroll the loop to see that there isn't a dependence:

```
i==0, x[3] = x[0] * 5
i==1, x[5] = x[2] * 5
i==2, x[7] = x[4] * 5
…
i==99, x[201] = x[198] * 5;
```

The GCD test is sufficient to guarantee that no dependence exists, but there are cases where the GCD test succeeds but there aren't any dependences. The GCD test doesn't look at array bounds, for instance.

In general, determining whether a dependence exists is NP-complete. However, there are many common cases that can be analyzed at low cost. There has been research that has shown that a hierarchy of exact tests that increase in generality do work on restricted cases to determine exactly whether there are dependencies.

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence, too. The classification allows the compiler to recognize if there are name dependencies, which can be eliminated at compile time by renaming and copying.

Example:

The following loop has multiple types of dependencies. Find all the true dependencies, output dependencies, and antidependences, and eliminate the output dependences and antidependences by renaming.

A “true dependency” is one that cannot be changed by renaming, although it may or may not limit the ability to parallelize (if it is loop carried, for instance).

An “antidependency” occurs when one instruction writes to a register or memory location that another instruction reads. The ordering is important for antidependences.

An “output dependency” occurs when two instructions write to the same register or memory location. Again, the ordering must be preserved to ensure that the value that is finally written is the one that happens last.

```
for (i=0; i<100; i=i+1) {
      Y[i] = X[i] / c; /* S1 */
      X[i] = X[i] + c; /* S2 */
      Z[i] = Y[i] + c; /* S3 */
      Y[i] = c - Y[i]; /* S4 */
}
```

Answer:

1. There are true dependencies from S1 to S3 and from S1 to S4 because of Y[i]. These aren’t loop carried, so they do not prevent the loop from being parallelized. They will, however, force S3 and S4 to wait for S1 to complete.

2. There is an antidependence from S1 to S2, based on X[i]

3. There is an antidependence from S4 to S4 for Y[i]

4. There is an output dependence from S1 to S4, based on Y[i]

We can eliminate the pseudo-dependencies with the following version of the loop:

```
for (i=0; i<100; i=i+1) {
      T[i] = X[i] / c; /* Y renamed to T to remove output dependence */
      X1[i] = X[i] + c; /* X renamed to X1 to remove antidependence */
      Z[i] = T[i] + c; /* Y renamed to T to remove antidependence */
      Y[i] = c - T[i];
}
```

After the loop, X has been renamed to X1. In the code that follows the loop, the compiler can simply replace the name X with X1. In this case, renaming doesn’t require copying, because we can substitute names, or by register allocation. In the other cases, though, we do need to rename by copying (e.g., T must be a copy of Y).

Eliminating Dependent Computations

Take a look at a common recurrence, the dot product:

```
for (i=9999; i>=0; i=i-1) {
      sum = sum + x[i] * y[i];
}
```

This loop isn't parallel because there is a loop-carried dependence on the variable sum. We can transform it to a set of loops, to remove the dependence. One loop will be completely parallel, and the other loop will be partially parallel.

Here is the completely parallel loop:

```
for (i=9999; i>=0; i=i-1) {
      sum[i] = x[i] * y[i];
}
```

The sum variable is now a vector (array) of its own, but each partial sum is independent.

But, now we need to reduce this sum into a scalar value again:

```
for (i=9999; i>=0; i=i-1) {
      finalsum = finalsum + sum[i];
}
```

This is just a reduction, which we have looked at before! While not completely parallel, it can be partially parallelized, and still run with multiple processors. E.g., with 10 processors, each with their own processor number, p:

```
for (i=999; i>=0; i=i-1) {
      finalsum[p] = finalsum[p] * sum[i+1000*p];
}
```

Now, we only need to sum up ten values at the end (or we could further parallelize that, with diminishing returns).

The one caveat to all of this: the transformation above relies on associativity of addition (i.e., the order of the addition operation doesn't matter). This is true for numbers with unlimited range and precision, but not necessarily true on computers! In fact, a reduction done on a GPU with floating point values might be slightly different than the result done on a CPU in serial. This can make debugging particularly hard!